Crear proyecto nest nestjs-products-api

[https://youtu.be/jEKsD5f3Bqc](https://www.youtube.com/watch?v=jEKsD5f3Bqc)

<https://www.youtube.com/watch?v=AmF_BTzJdFY>

Crear el proyecto con el comando

nest new nestjs-products-api

generar un repositorio a traves de GIT

git init

crear el archivo gitignbore en la raíz del proyecto y agregar node\_modules para que ignore en el repositorio lo que es la carpeta con las dependencias.

Generar el modulo product

nest g module product

lo que creara la carpeta src/product y el product.module.ts, entonces continuamos con

nest g controller product

nest g service product

Cremos una carpeta **/dto** y dentro un archivo **product.dto.ts** donde se definirá lo que se estará enviando entre el cliente (browser) y el servidor.

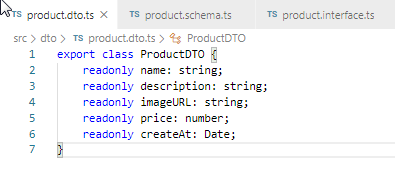
Creamos otra carpeta **/interfaces** y dentro un archivo **product.interface.ts** donde se definirán las interfaces que ayudaran a describir nuestros datos, para que typescript nos brinde ayuda en el autocompletado,el análisis de errores, etc.

Vamos a necesitar guardar datos y para esto vamos a necesitar una BD. Vamos a utilizar MongoDB, en este caso no van a existir esquemas (un modelo de datos), en MongoDB se guardan directamente los objetos.

Vamos a crear una carpeta **/schemas** que está más relacionada con MongoDB que con Nestjs ya que se va a definir lo que se va a guardar en MongoDB. Alli creamos un archivo **product.schema.ts**.

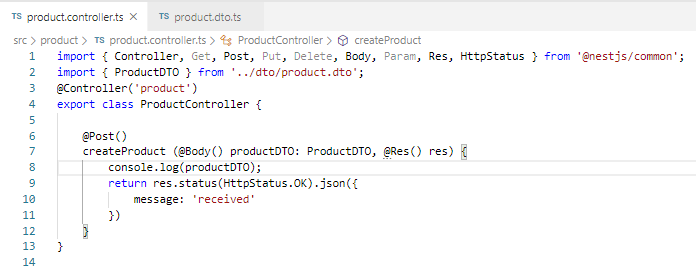
Ya podemos arranar nuestro servidor.

Vamos a definir en product.dto.ts la clase ProductDTO que será de la siguiente manera

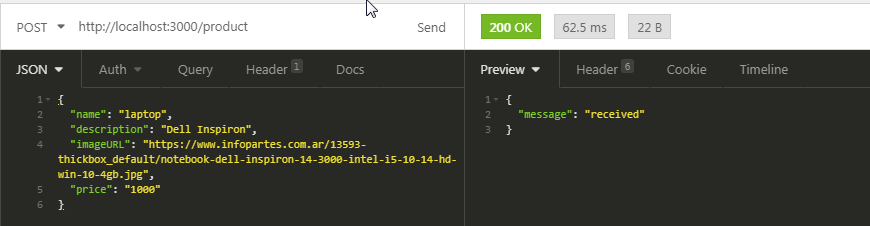


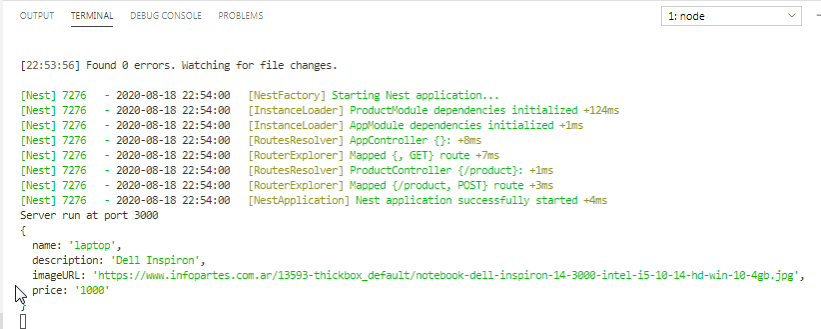
Dentro del controlador de producto product.controller.ts vamos a crear las rutas para GET, POST, PUT y DELETE.

Arranquemos por el POST para crear el producto



Aquí ya podemos hacer las primeras pruebas del servidor y la ruta del POST





Agreguemos lo necesario para MongoDB, neceistare intaslar Mongoose que es un ODM (Object-Document Model – lo que el ORM es a las BD Relacionales) para MongoDB y además debo instalar la integración de Mongoose con Nestjs

npm install @nestjs/mongoose mongoose

también vamos a instalar @types/mongoose un modulo de typescript con integración a mongoose que ayuda en el reconocimiento de la sintaxis de mongoose

npm install @types/mongoose

y ahora hay que configurarlo. Cuando es necesario configurar algo en un librería (modulo) agregado esto se hace en el arreglo import: [] del archivo module.ts, y como la conexión a la BD es algo global a la aplicación, por esto es que se hará en el app.module.ts. Importo MongooseModule de @nestjs/mongoose y lo agrego al import pero además hay que indicarle la cadena de conexión mediante el método forRoot(), de la siguiente manera



Para poder probar debemos levantar el motor de la BD mongoDB, esto lo hacemos ejecutando el comando **Start\_mongo\_Service.cmd** que indica lo siguiente

mongod.exe --dbpath c:\mongodb\data\db

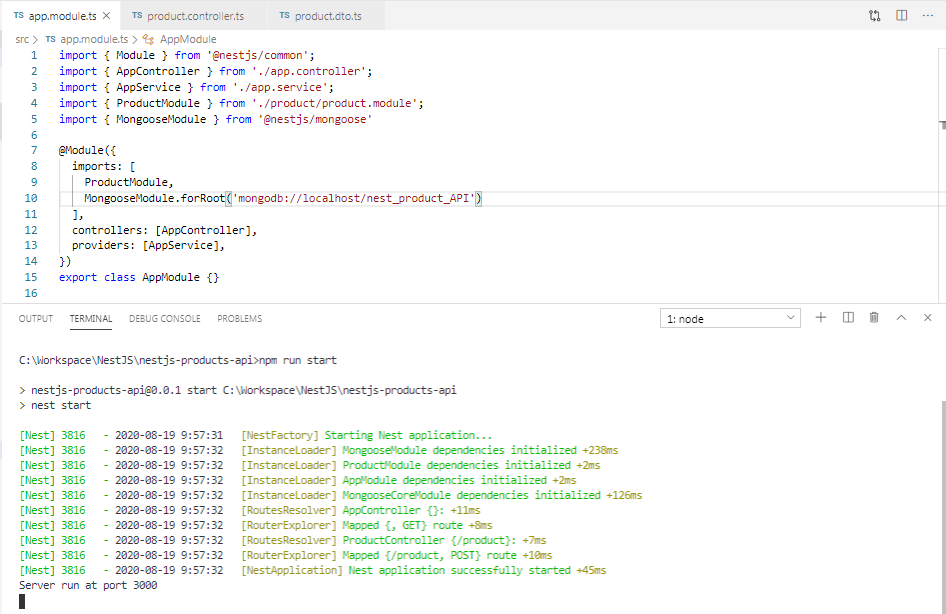
**NOTA**: si al intentar levantar el servidor diera algún error en la conexión a la BD, quiza necesite agregar una configuración más en la conexión

    MongooseModule.forRoot('mongodb://localhost/nest\_product\_API', {

      useNewUrlParser: true

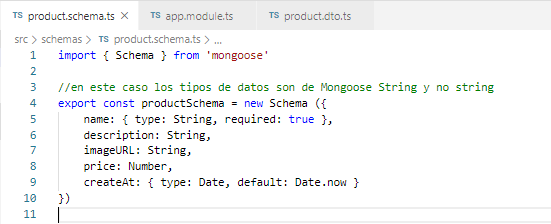
    })

En mi caso no fue necesario.



Para empezar a definir los métodos para leer y guardar datos en la BD vamos a hacerlo en el **product.service.ts**.

Pero antes deberemos definir el **schema** que va a vincular los datos en la aplicación con la BD y la **Interface**.

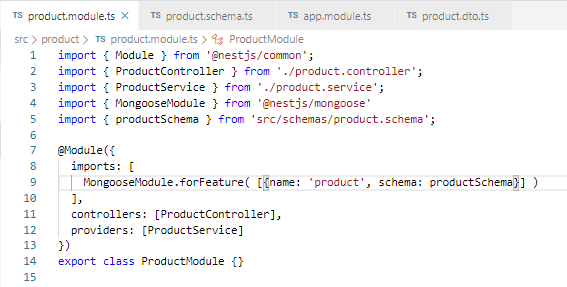


por ahora, en nuestra aplicación solo tenemos el esquema de los productos, pero en una aplicación real seguramente va a haber más.

Pero con el esquema solo no basta para guardar datos, para esto necesito un **Modelo**, el esquema solo define los datos, necesitaremos utilizar el modelo para guardar los datos en la BD MongoDB.

Dentro del modulo product, más precisamente en el archivo **product.module.ts** y puntualmente en el apartado **import** es donde debemos definir estas particularidades del modulo. Anteriormente con la conexión a la BD lo hicimos dentro del **app.module.ts** porque lo referente a la conexión es global a la aplicación pero en este caso el esquema del producto es particular al producto, por esto se hace en este módulo.

Previamente importamos el modelo **MongooseModule** de **@nestjs/mongoose** y de este utilizamos el método **forFeatue()** para registrar el esquema **productSchema**



Dentro de los **imports** del **ProductModule** mediante el método **forFeature** del **MongooseModule** el esquema **productSchema** con nombre product(lo defino dentro de un arreglo [] pues puede que haber más esquemas)

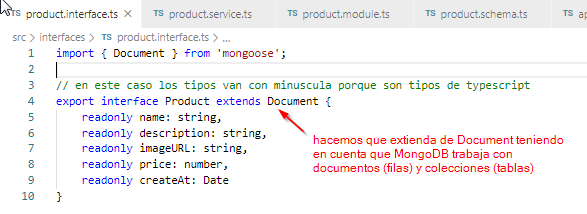
  imports: [

    MongooseModule.forFeature( [{name: 'product', schema: productSchema}] )

  ],

Luego en el ProductService lo inyectaremos en el modelo.

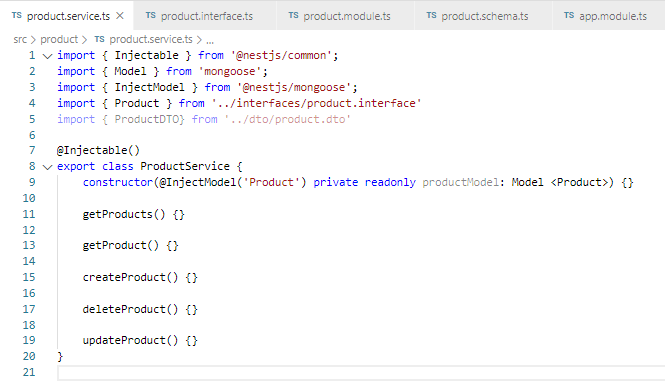
Ahora definimos y exportamos la interface **Product** en el archivo **product.interface.ts** , es muy similar al DTO solo que la interfaz detallará lo que se está moviendo dentro del código (el dto detalla lo que se mueve entre cliente y servidor).



En este caso tener en cuenta que los tipos son de Typescript y allí son con minúscula (string, number), Date si es con mayúscula.

Ahora si en el **servicio** hacemos los **imports** necesarios y dentro del **constructor** inyectamos el **modelo** que indicamos que es **Product** (asi se lo definió en el modulo **product.module.ts**) le damos el nombre de **productModule** y decimos que éste es del tipo **Model** (importado de **mongoose**) y a su vez aclaramos que implementa la **interfaz Product**.

Y por otro lado podemos definir los métodos que va a implementar la clase ProductService, uno para leer todos los productos, otro para encontrar un producto otro para dar de alta un producto, otro para eliminar y finalmente uno para modificar. Los podemos escribir sin código inicialmente para ir arrancando de a poco.



Mongoose cuenta con diferentes métodos para realizar las tareas mencionadas. El método

**find()** retornara un arreglo de todos los objetos Product

**findById(\_id)** retornara el producto con un \_id determinado o un undefiend

**save()** permite crear un documento nuevo en la BD

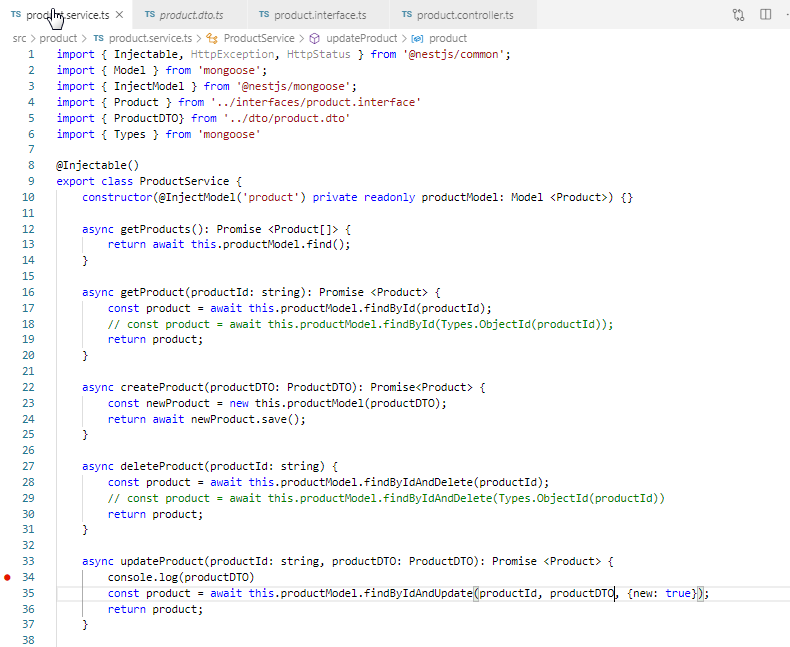
**findByIdAndDelete(\_id)** elimina un producto y retorna el producto eliminado o undefined si no encontró in producto con dicho \_id

**findByIdAndUpdate(\_id, object, [options])** en este caso actualiza el producto si encuentra uno con ese \_id, sino retorna undefined. Donde **object** puede ser

* el objectDTO enviado en el **body** (como un json) con todas las propiedades (las que se modifican y las que no)
* puede ser un json del tipo {operator : {propiedad : valor}}

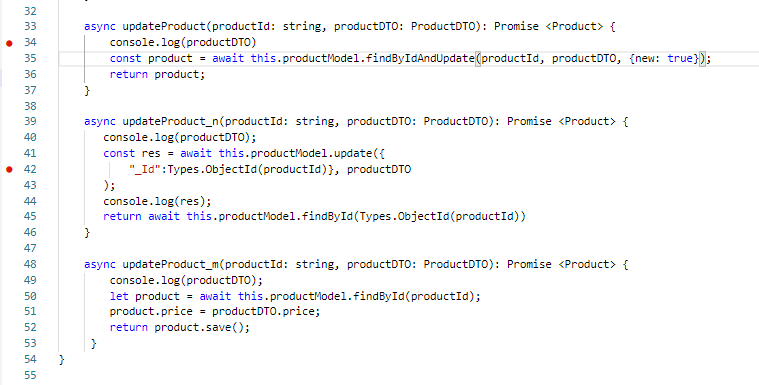
por ejemplo el operador por default es $set pero hay varias opciones para utilizar ve la documentación de la función en la API Mongoose

**options**, es opcional, tiene la forma de objeto {opción: valor} ver la documentación de la función. Una de las opciones la cual usamos es {new: ‘true} esto es para decirnos que nos retorne el producto actualizado, por default new esta seteado en false y esto hace que la función retorne el producto con los valoras previos a la actualización



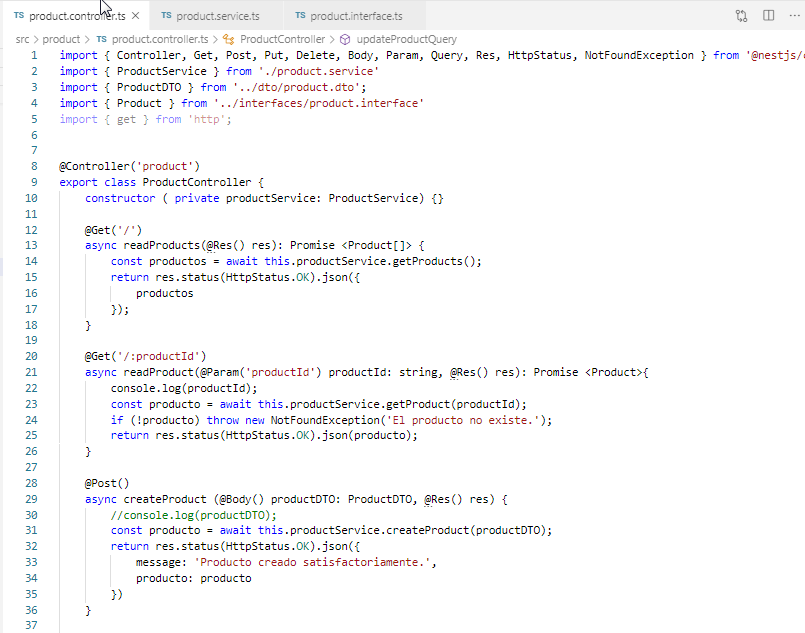
Otro punto importante con este último método, es que la operación es atómica. Se podría pensar en hacer una actualización haciendo búsqueda mediante **findById()** del producto, modificarlo y luego aplicarle el método **save()**… pero ahí perdemos atomicidad y habría que comenzar a considerar concepto de transacción.

Siempre que haceos un **find()** usamos para almacenar los retornos una variable de tipo de la interfaz **Product**, el problema que allí existe es que las propiedades están definidas como **readonly** entonces cuando las intento actualizar sobre el objeto encontrado me da error, habría que quitarle el **readonly** o definir otra para estos casos.



Por otro lado existen otros métodos para hacer actualizaciones como **findOneAndUpdate()**, **Update()**, **UpdateOne()** **updateMany()** **findOneAndDelete()**. Lo mismo para eliminar **findByIdAndDelete()**, **findOneAndDelete()**.

El controlador **product.controller.ts** contendrsa las llamadas a estos métodos. Dentro del constructor de la clase **ProductController** iniyectamos objeto **productSevice** que será sobre el cual se trabajara para las diferentes rutas (endpoints) y los diferentes métodos HTTP



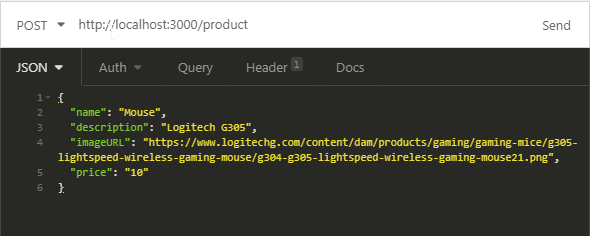
Tenemos dos **GET** uno sin parámetros que traerá todos los productos de la BD, el otro con el parámetro de productId.

En el del último caso tenemos la posibilidad de usar uno de dos decoradores **Param** o **Query** y según sea el caso la ruta para la llamada podrá ser

@Param(‘productId’) 🡪 <http://localhost:3000/product/5f43160d4e12f20608cd37c7>

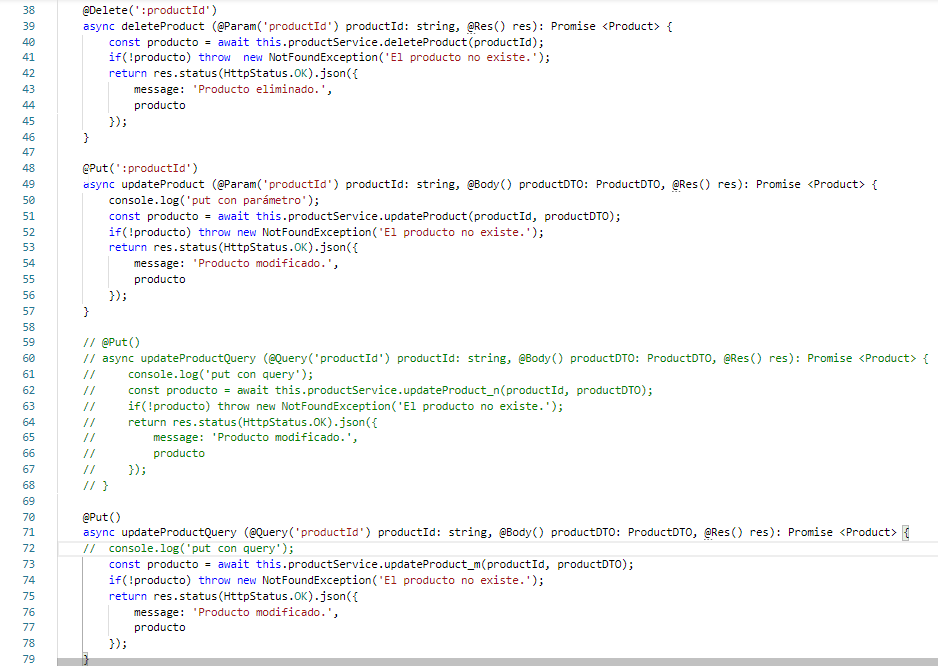
@Query(‘productId’) 🡪 <http://localhost:3000/product?productId=5f43160d4e12f20608cd37c7>

Para el caso del **POST** usare un decorador **Body** y lo inicializare con un json desde Insomnia o postman



Para el **DELETE** también requeriré indicar cual eliminar mediante el parámetro productId, y puede pasarse como **Param** o como **Query** como se hico en el **GET**.

En el caso de **PUT** paso el parámetro productId, también como **Param** o como **Query** y uso un **Body** para pasar un **productDTO** que cargo desde un json desde Insomnia o Postman



En el caso del **updateProduct()** hice tres diferentes para probar con el pase de **Param** o **Query** y para usar un método con **findByIdAndUpdate()** y orto con **FindById()** y **save()**